TypePlug -- Practical, Pluggable Types

Nik Haldiman
Marcus Denker
Oscar Nierstrasz

University of Bern

Types?

Static typing is Good!

- Programs with failures are rejected
  - Reduces errors detected at runtime
- Documentation
- Minor inconvenience, major payoff

Static typing is Evil!

- Exactly all cool programs are rejected
  - Reflection?!
- Inconvenience is not at all “minor”
  - Typed programs hard to change + evolve
- Only the most trivial errors are detected
  - False sense of security

Cakes by Darcy

Is it possible to have one's cake and eat it, too?

Pluggable Types

- Optional: does not change the semantics

- Pluggable: many different ones
  - Especially exotic type-systems

- “Type-Systems as Tools”

Gilad Bracha, OOPSLA 04:
Pluggable Type-Systems

The Problem

UNIVERSITÄT BERN

- Large, untyped code-base
- Overhead for using pluggable types is high
  - Existing code needs to be annotated with type information

TypePlug

- Pluggable types for Squeak
- Based on sub-method reflection framework (Demo on Wednesday!)
- Case-Studies:
  - Non-Nil Types
  - Class Based Types
  - Confined Types

Non-Nil Type-System

UNIVERSITÄT BERN

> Declare variables to never be nil

```
Object subclass: #Line
   typedInstanceVariables: ’startPoint endPoint <:nonNil:>’
   typedClassVariables: ’’
   poolDictionaries: ''
   category: 'Demo'
```

DEMO

Non-Nil Type-System

```
moveHorizontally: anInteger

   startPoint := self movePoint: startPoint
                     horizontally: anInteger.

   endPoint:=self movePoint: endPoint
                 horizontally: anInteger
```

Non-Nil Type-System

```
moveHorizontally: anInteger

   startPoint := self movePoint: startPoint
                      horizontally: anInteger.

   endPoint:=self movePoint: endPoint
                 horizontally: anInteger <- type 'TopType' of
   expression is not compatible with type 'nonNil' of variable
   'endPoint'.
```

Non-Nil Type-System

```
movePoint: aPoint horizontally: anInteger

             ↑ (aPoint addX: anInteger y: 0) <:nonNil :>
```

The Problem (again)

> Large, untyped code-base

> Overhead for using pluggable types is high

  — Existing code needs to be annotated with type information

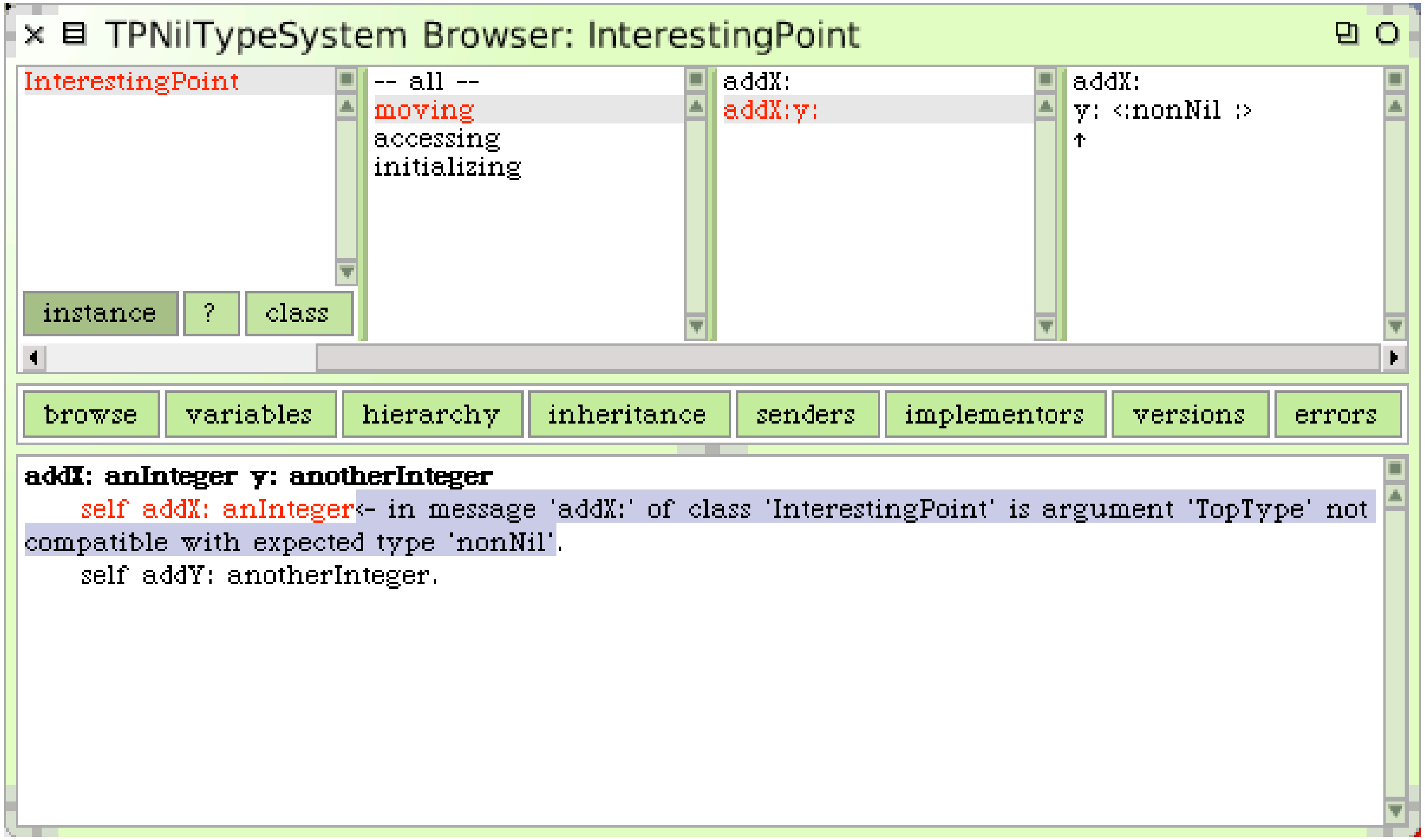
Solution

- Only type-check annotated code
- Use type-inference to infer types of non-annotated code
- Explicit type-casts
- Allow external annotations for foreign code

External Type Annotations

- We need to annotate existing code
  - Especially libraries and frameworks
  - Example: Object>>#hash is <: nonNil :>
- We do not want to change the program code!
- Solution: External Type Annotations
  - Added and modified in the TypesBrowser
  - Do not change the source
  - External representation: Type Packages

Browser



TPNilTypeSystem Browser: InterestingPoint

```
addX:
y: <:nonNil :>
↑
```

```
addX: anInteger y: anotherInteger
    self addX: anInteger<- in message 'addX:' of class 'InterestingPoint' is argument 'TopType' not compatible with expected type 'nonNil'.
    self addY: anotherInteger.
```

Future Work

- Improve Type-Inference
  - Better algorithms
  - Explore heuristical type inference (Roeltyper)

- Type Checking and Reflection
  - Use pluggable types to check reflective change

Conclusion

> TypePlug: Pragmatic framework for Pluggable Types

- Only type-check annotated code
- Use type-inference
- Explicit type-casts
- External annotations for foreign code

Conclusion

> TypePlug: Pragmatic framework for Pluggable Types

- Only type-check annotated code
- Use type-inference
- Explicit type-casts
- External annotations for foreign code

Questions?